Part 3: Asymmetric Cryptography and Certificates

In this part, we will learn about asymmetric cryptography, public and private encryption keys, digital signatures and their verification, and X.509 certificates. This part will compare asymmetric ciphers, such as RSA and Elliptic Curves. All the technologies mentioned will be illustrated using command-line and C code examples.

This part the following chapters:

- *Chapter 6, Asymmetric Encryption and Decryption*
- *Chapter 7, Digital Signatures and Their Verification*
- *Chapter 8, X.509 Certificates and PKI*

6
Asymmetric Encryption and Decryption

In this chapter, we will learn about **asymmetric encryption**, how it works, and how public and private keys are used to achieve encryption and decryption. In the practical part of this chapter, we will learn how to use asymmetric encryption and decryption on the command line and in C code.

We are going to cover the following topics in this chapter:

- Understanding asymmetric encryption
- Understanding a Man in the Middle attack
- What kind of asymmetric encryption is available in OpenSSL?
- Understanding a session key
- Understanding RSA security
- How to generate an RSA keypair
- How to encrypt and decrypt with RSA on the command line
- How to encrypt with RSA programmatically
- Understanding the OpenSSL error queue
- How to decrypt with RSA programmatically

Technical requirements

This chapter will contain commands that you can run on a command line and C source code that you can build and run. For the command line commands, you will need the `openssl` command-line tool with OpenSSL dynamic libraries. For building the C code, you will need OpenSSL dynamic or static libraries, library headers, a C compiler, and a linker.

We will implement some example programs in this chapter, in order to practice what we are learning. The full source code of those programs can be found here: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter06`

Understanding asymmetric encryption

As we learned in *Chapter 2*, *Symmetric Encryption and Decryption*, symmetric encryption algorithms use the same key for encryption and decryption. **Asymmetric encryption algorithms**, conversely, use two keys – **a public key** and **a private key**. A public key and its corresponding private key form a **keypair**. The public key is used for encryption and the private key is used for decryption.

Why do we need the complication with two keys? Why can't we just always use symmetric encryption with one key? In short, asymmetric encryption is needed when it is difficult or impossible to send a secret key and ensure that no one eavesdropped on the transferred key. Let's imagine that Alice wants to send a message to Bob over a non-secure channel – for example, over the internet. On the internet, transit hosts between Alice and Bob can eavesdrop on the passing traffic and even change it. Alice does not want anyone to eavesdrop on the message, therefore she decides to encrypt the message. But how to transfer the encryption key? If Alice and Bob had a freely available secure communication channel for sending the key, they could just send the actual message over that channel, without encryption. Thus, Alice and Bob decide to use asymmetric cryptography:

1. Bob generates a keypair and sends his public key to Alice.
2. Alice encrypts the message with Bob's public key and sends the encrypted message to Bob.
3. Bob decrypts the message with his private key.

If an attacker eavesdrops on the traffic and gets both Bob's public key and the message, they will not be able to decrypt the message. Only Bob's private key can decrypt the message. Therefore, Bob has to keep his private key secret and not share it with anyone.

It works well if a potential attacker can only passively eavesdrop on the traffic but cannot change it. But what if they can? Then, they can attempt a Man in the Middle attack.

Understanding a Man in the Middle attack

A **Man in the Middle** (**MITM**) attack is an attack where an attacker both listens in on the transit traffic and changes it, trying to impersonate the receiver to the sender and the sender to the receiver.

Let's demonstrate a possible attack for the scenario with Alice and Bob mentioned previously. Let's suppose that Mallory acts as a Man in the Middle in order to recover the plaintext of the encrypted message that Alice wants to send to Bob. Then, the attack scenario will be as follows:

1. Bob generates a keypair and sends his public key to Alice.

2. Mallory generates her own keypair. She intercepts Bob's public key sent to Alice and saves it for future use. Instead of Bob's public key, Mallory sends her own public key to Alice, disguised as Bob's key.
3. Alice encrypts her message with Mallory's public key, thinking that it is Bob's public key. Alice then sends the encrypted message to Bob.
4. Mallory intercepts Alice's message. Mallory decrypts the message with her private key. Then, she re-encrypts the message with Bob's public key and sends the encrypted message to Bob.
5. The outcome is that Alice thinks that she sent the message to Bob and no one else has read it. Bob thinks that he received the message from Alice and no one else has read it. Mallory got away with the secret information from the message, unnoticed.

How do we defend against an MITM attack? There are different methods, but each method requires a trusted communication channel available at least for a short time before the main communication happens.

Meeting in person

The two parties (Alice and Bob) can meet in person and exchange or verify their public keys. This method is trivial but often not very practical, especially if Alice and Bob live far away from each other. However, sometimes it is practical. For instance, some IT conferences include **key signing parties**. People who come to the conference from distant parts of the world gather and show each other IDs, such as passports or driving licenses, and their **Pretty Good Privacy** (**PGP**) key fingerprints. Participants of the party can also sign each other's keys; that's why it is called a key signing party.

Verifying a key fingerprint over the phone

A public key fingerprint is basically its full or partial cryptographic hash, hex- or base64-encoded. After exchanging the public keys over the internet – for example, via email – Alice and Bob have a phone call and verify each other's fingerprints. In such a case, Alice and Bob's voices serve as a form of authentication. Unfortunately, nowadays, this key verification method is less practical and reliable than before. Faster CPUs and advanced cryptanalysis methods force Alice and Bob to use longer fingerprints, and recent developments of deepfake technologies make it possible to falsify human voices in real time. Even so, verification by voice is still quite a good authentication method. In order to falsify the key exchange, an attacker needs to accomplish an MITM attack on two different communication channels and employ advanced deepfake technology.

Key splitting

The key can be split into several parts, and each part can be sent to the recipient using a different delivery method, such as email, instant messenger, publishing on a file exchange server, and sending a flash drive by physical mail. The idea is that it will be difficult for an attacker to intercept and falsify messages on all channels. After the recipient has reassembled the key, its fingerprint can additionally be verified over the phone.

Signing the key by a trusted third party

A third party – let's call him Thomas – who is trusted by both Alice and Bob can digitally sign Bob's public key, certifying that it really is Bob's key. Then, Alice can verify the signature. This method requires that Alice must first get Thomas' public key via some trusted channel. A similar but more complex model of key verification is used in PGP's **web of trust** and in **Public Key Infrastructure** (**PKI**), based on X.509 certificates. More information about digital signatures is available in *Chapter 7*, *Digital Signatures and Their Verification*, and more details about X.509 certificates can be found in *Chapter 8*, *X.509 Certificates and PKI*.

Alice and Bob now know how to securely exchange the public keys. But what asymmetric encryption algorithm should they use in order to encrypt the messages that they want to securely send to each other? Let's learn about that in the next section.

What kind of asymmetric encryption is available in OpenSSL?

The OpenSSL library implements several asymmetric crypto algorithms, but only one of those algorithms allows you to *directly* encrypt data. It is the **Rivest-Shamir-Adleman** (**RSA**) algorithm.

Other available asymmetric crypto algorithms, such as **Digital Signature Algorithm** (**DSA**) and **Elliptic Curve Digital Signature Algorithm** (**ECDSA**), can be used for digital signatures. **Diffie–Hellman** (**DH**) and **Elliptic Curve Diffie–Hellman** (**ECDH**) algorithms can be used for key exchange in the **Transport Layer Security** (**TLS**) protocol.

As we know, symmetric cryptography keys do not have structure; they are just arrays of random bits. Conversely, asymmetric crypto algorithms use structured keys, meaning that a key can have several components, and a component may have certain requirements – for example, it must be a prime number instead of a random bit array. Each asymmetric crypto algorithm has its own structure of public and private keys. Hence, there are different formats of asymmetric keys, such as RSA keys and ECDSA keys.

Why did we say previously that RSA is the only asymmetric crypto algorithm available in OpenSSL that can *directly* encrypt data? What does *directly* mean? It means that the RSA algorithm alone is enough for encryption. Can other algorithms be used *indirectly* for encryption? Yes, it is possible. There

is the **ElGamal** algorithm, which can use a DH algorithm with DSA keys, or an ECDH algorithm with ECDSA keys, in order to do asymmetric encryption. ElGamal involves creating an ephemeral (temporary) DSA/ECDSA key in addition to the DSA/ECDSA key that you want to use for encryption, doing a DH/ECDH key exchange between the two keys in order to get the shared secret, deriving a symmetric encryption key from the shared secret, and finally, encrypting the data with that symmetric key. As we can see, it is more complicated than using RSA. There are also other schemes for deriving a symmetric key using a defined asymmetric key and an ephemeral asymmetric key, such as **Integrated Encryption Scheme** (**IES**) and **Hybrid Public Key Encryption** (**HPKE**). OpenSSL does not implement ElGamal, IES, HPKE, or similar algorithms. Therefore, for asymmetric encryption examples in this chapter, we will use RSA.

A symmetric encryption key used in an ElGamal algorithm is called a **session key**. The concept of using a session key is not specific to ElGamal. A session key is also used with RSA and in secure network protocols, such as TLS, SSH, and IPsec. Let's look at the session key in detail next.

Understanding a session key

It is important to understand how asymmetric encryption is used in practice. Asymmetric ciphers, such as RSA, are much slower than symmetric ciphers, such as AES. Therefore, usually, the actual data that a sender wants to encrypt is not encrypted by RSA. Instead, the sender generates a symmetric session key. Then, the actual data is encrypted by a symmetric algorithm, such as AES, with the session key, which is encrypted by RSA. When decrypting, the recipient first decrypts the session key by RSA and then decrypts the actual data by AES with the session key. Such an encryption scheme is often referred to as a **hybrid encryption scheme**.

In secure network protocols, such as TLS, SSH, and IPsec, the communication session begins with the **handshaking** operation, part of which is the **key exchange** operation. In the older versions of the protocols, the key exchange operation involved the generation of the session key by one party of the communication, encrypting it by RSA, and sending it encrypted to the other party. In the current versions of the protocols, the communication parties use the DH or ECDH key exchange methods, where both communication parties derive the same session key, which they then use for encryption of the useful data. More information about TLS handshaking will be available in *Chapter 9*, *Establishing TLS Connections and Sending Data over Them*.

For the hybrid encryption to be secure, both symmetric encryption by the session key and asymmetric encryption by the RSA key must be secure. The next section gives an insight into RSA security.

Understanding RSA security

The security of RSA relies on the difficulty of the integer factorization problem. As a reminder, the factorization operation is breaking a positive integer number into a multiplication of prime numbers. Currently, there is no polynomial-time algorithm for integer factorization that can be run by a classical computer, and hence, this problem remains very computationally expensive. And how big are the integer

numbers used in RSA? Typically, they are hundreds or thousands of decimal digits, or thousands or tens of thousands of bits. Apparently, usual integer types in C language are not large enough to perform mathematics with big numbers used in RSA. Hence, OpenSSL includes a big-number sub-library in order to deal with big numbers. Types and functions of that big-number sub-library have a `BN_` prefix.

An RSA public or private key consists of several big or small integer numbers called a modulus, an exponent, and, optionally, primes and coefficients. As mentioned, one of those numbers is a modulus. The RSA key size is considered to be the size of the modulus. For instance, if an RSA key contains a 2,048-bit modulus, then the size of that RSA key is also 2,048 bits.

It's important to remember that the security level of an RSA key is much lower than the key size. For example, a 2,048-bit RSA key has a security level of only 112 bits. The security level of an RSA key does not grow linearly depending on the key size. The key size must be increased substantially in order to gain only a few security bits.

Here is a table that lists security levels estimated by the National Institute of Standards and Technology (NIST) for RSA keys of different sizes:

RSA key size in bits	Security level in bits
1,024	80
2,048	112
3,072	128
4,096	152
7,680	192
8,192	200
15,360	256

Table 6.1: Security levels for different RSA key sizes

As we can see, in order to reach the same security level as AES-256, we need an RSA key of 15,360 bits! Are there any disadvantages with long RSA keys? Unfortunately, there are:

- There is slower performance for both encryption and decryption, and signing and signature verification. Performance drops faster than linear with the growth of the key size. When doubling the key size in the range of 1024–4096 bits – for example, increasing the key size from 1,024 to 2,048 bits – encryption and signing becomes approximately 7 times slower, and decryption and signature verification becomes approximately 3 times slower. To give a rough idea about RSA speed, a PC with anIntel i5 CPU can do thousands of RSA encryptions or hundreds of RSA decryptions with a 4,096-bit key per second. As we can see, RSA encryption is much faster than RSA decryption. If you're curious, you can check speeds on your machine using the `openssl speed rsa` command.

- RSA outputs encrypted data in blocks equal to the key size. Considering that RSA is usually used for the encryption of a symmetric key and possibly an initialization vector, RSA encryption will expand the input data several times – for example, a 256-bit symmetric key and a 128-bit initialization vector encrypted by a 2,048-bit RSA key will produce 2,048-bit-long ciphertext. The longer the RSA key, the more length expansion during the RSA encryption will happen. Technically, the result of the RSA encryption is a very long integer, thus it is possible to strip leading zeros from it and shorten the ciphertext a little bit. But the benefit is not worth the hassle, and no popular encryption software implements such shortening.
- The RSA signature size in the standard PKCS #1 format is also equal to the RSA key size. Longer signatures are inconvenient.
- Longer RSA keys take a longer time to be generated – for example, on a PC with Intel i5 CPU, a 4,096-bit RSA key is generated in less than a second, but the generation of a 15,360-bit key can take 2 minutes.

As we can see, we have a trade-off between security and performance. The amount of space that an RSA key, encrypted block, or signature takes is much less of a concern. Unfortunately, in order to get only a few more bits of security, we have to increase the key size by many more bits, and we have to pay a significant performance penalty. Fortunately, CPUs, including those in mobile and embedded devices, are becoming faster and faster.

But what is the acceptable compromise? What RSA key size do we choose? NIST recommends using the following sizes for RSA keys:

- At least 2,048 bits until 2030
- At least 3,072 bits after 2030

At the time of writing, 2,048 bits is both the most popular and minimally acceptable size for RSA keys. Most RSA-based web server TLS certificates on the web use 2,048-bit keys. The next step will likely be 4,096-bit certificates. Many old articles on the internet advocated a compromise of 3,072-bit keys and certificates, but that idea didn't really take off. Current articles usually compare 2,048-bit and 4,096-bit certificates.

I personally think that it is nice to have some security margin, thus I prefer 4,096 bits as the RSA key size. Why not less? Because computers that I use work comfortably fast with such RSA keys. Why not more? Because 4,096 bits provide enough of a security margin and because some software, especially old software, does not support RSA keys longer than 4,096 bits. For example, even modern GnuPG does not allow you to create an RSA keypair longer than 4,096 bits. Also, for what it's worth, 4,096 is a nice number, because it is a power of 2, unlike 3,072.

I also think that it is even better to use **Elliptic Curve Cryptography** (**ECC**) instead of RSA in cases where the usage of ECC is as easy as the usage of RSA – for example, in TLS certificates and SSH keys. But for encryption, RSA is much more straightforward to use than ECC. We will learn more about ECC in *Chapter 7*, *Digital Signatures and Their Verification*.

What will happen with RSA security when **quantum computing** arrives? Unfortunately, asymmetric cryptography as we know it, both RSA and ECC, will be completely broken by quantum computing, and we don't have a standardized and practice-proven replacement yet. The good news is that many cryptographers are actively working on post-quantum cryptographic algorithms. At the time of this writing, NIST is running post-quantum cryptography standardization, which is a competition for quantum-resistant crypto algorithms, similar to AES and SHA-3 competitions. The winners of the competition will be standardized as the standard post-quantum cryptography algorithms. The most optimistic predictions are that quantum computers capable of breaking the current asymmetric cryptography will be created in the 2030s. Thus, we have a good hope that post-quantum crypto algorithms will be standardized and included in popular crypto libraries, such as OpenSSL, before powerful quantum computers become available even to big players, such as governments and corporations.

Quantum computing and OpenSSL support for post-quantum crypto algorithms are things that we expect in the future. But what can we do now? Learn how to use RSA. Let's start with generating our first RSA keypair.

How to generate an RSA keypair

The `openssl` tool provides two subcommands for generating RSA keypairs – `genrsa` and `genpkey`. The former can generate only an RSA keypair, while the latter is a more generic subcommand that can generate any type of keypair supported by OpenSSL. `genrsa` is declared deprecated since OpenSSL 3.0, thus we will use `genpkey`.

Documentation for the `openssl genpkey` subcommand can be found on the `openssl-genpkey` man page:

```
man openssl-genpkey
```

Why such a name, `genpkey`? OpenSSL has a concept of a **Public or Private Key** (**PKEY**). Here, it is important to clear up one confusion. Throughout the OpenSSL documentation, you will find mentions about public and private keys. Very often when mentioning a private key, the documentation really means a keypair. It applies to both command-line tools documentation and OpenSSL API documentation. For example, the `description` part of the `openssl-genpkey` man page says, *The genpkey command generates a private key.* If only a private key is generated, how would I generate the complementary public key? The OpenSSL documentation suggests that I can extract a public key from a private key, which can be quite confusing for inexperienced OpenSSL users. More specifically, what OpenSSL and PKCS standards usually call a private key is a data structure, holding information sufficient for constructing both private and public keys. Some of that information is shared between the public and the private key, particularly the modulus part for RSA. A public key is a similar data structure, containing only public key information. Therefore, for simplicity, we can assume that when mentioning a private key, OpenSSL means a keypair, and when mentioning a public key, OpenSSL means what it says – a public key.

I hope I was able to clear up more confusion than I created! Now, let's generate our first RSA keypair:

```
$ openssl genpkey \
    -algorithm RSA \
    -pkeyopt rsa_keygen_bits:4096 \
    -out rsa_keypair.pem
```

Using the `-pkeyopt rsa_keygen_bits:4096` switch, we have specified that we want to generate a 4,096-bit keypair. After the generation, the keypair has been written into the file named `rsa_keypair.pem`.

The keypair file can be opened with a text editor and will contain something like this:

```
-----BEGIN PRIVATE KEY-----
MIIJQwIBADANBgkqhkiG9w0BAQEFAASCCS0wggkpAgEAAoICAQDC2/
sxiM72yGgy
... a lot of base64-encoded data …
FZTCpfZK4ecBXkHHaVnHBmdS10EyngU=
-----END PRIVATE KEY-----
```

The keypair is saved in the **Privacy Enhanced Mail** (**PEM**) format. PEM is the default format used by OpenSSL for storing keys or certificates. Even though *mail* is mentioned in the PEM format name, the format is not just used for mail. PEM format is really a Base64 wrapping around some binary data, with a text header (the `BEGIN` line) and a text footer (the `END` line). If you remove the header and the footer from the keypair PEM file and Base64-decode it, you will get the keypair in the **Distinguished Encoding Rules** (**DER**) format. DER is another popular format for storing keys and certificates supported by OpenSSL. DER is a binary representation for data structures described by **Abstract Syntax Notation One** (**ASN.1**). ASN.1 is a notation that allows you to describe data structures for serialization in a platform-independent way – for example, an RSA keypair described by ASN.1 is an ASN.1 sequence of ASN.1 integers.

We can inspect the structure of the generated keypair using the `openssl pkey` subcommand. Documentation for the `openssl pkey` subcommand can be found on the `openssl-pkey` man page:

```
man openssl-pkey
```

There is also a similar RSA-specific subcommand, `openssl rsa`. It has been deprecated since OpenSSL 3.0, together with `openssl genrsa`, `openssl rsautl`, and other key type-specific subcommands.

Here is how we inspect the keypair structure:

```
$ openssl pkey -in rsa_keypair.pem -noout -text
Private-Key: (4096 bit, 2 primes)
```

```
modulus:
    …
publicExponent: 65537 (0x10001)
privateExponent:
    …
prime1:
    …
prime2:
    …
exponent1:
    …
exponent2:
    …
coefficient:
    …
```

It is very good that we have generated a keypair. Now, we have to extract the public key out of it in order to share the public key with a person that will encrypt data for us, because we should never share our keypair; we should only share the public key.

This is how we extract the public key from the keypair:

```
$ openssl pkey \
    -in rsa_keypair.pem \
    -pubout \
    -out rsa_public_key.pem
```

Note the `-pubout` switch. That's how we specified that we want only the public key in the output, not the whole keypair.

The resulting `rsa_public_key.pem` file does not contain secret information and can be freely shared. We can inspect the structure of our public key file:

```
$ openssl pkey -pubin -in rsa_public_key.pem -noout -text
Public-Key: (4096 bit)
Modulus:
    …
Exponent: 65537 (0x10001)
```

Note the `-pubin` switch. That's how we specified that the `openssl pkey` subcommand must treat the input as a public key, not as a keypair.

As we can observe, the public key file contains much less information than the keypair file – only the modulus and the public exponent.

We have spent enough time on key generation. Now, let's put those keys to some use.

How to encrypt and decrypt with RSA on the command line

The `openssl` tool provides two subcommands for encrypting with RSA – `pkeyutl` and the deprecated, RSA-specific `rsautl` subcommand. We will, of course, use `pkeyutl`. Documentation for that subcommand can be found on the `openssl-pkeyutl` man page:

```
man openssl-pkeyutl
```

As explained previously, RSA is usually used for encrypting a session key, which will then be used to encrypt useful data. Let's generate a 256-bit session key:

```
$ openssl rand -out session_key.bin 32
```

Now, let's use `openssl pkeyutl` with our public RSA key for encrypting the session key:

```
$ openssl pkeyutl \
    -encrypt \
    -in session_key.bin \
    -out session_key.bin.encrypted \
    -pubin \
    -inkey rsa_public_key.pem \
    -pkeyopt rsa_padding_mode:oaep
```

Note the `-pkeyopt rsa_padding_mode:oaep` switch. It instructs `openssl pkeyutl` to use the PKCS #1 v2.0 **Optimal Asymmetric Encryption Padding** (**OAEP**) padding type for the RSA encryption. It is strongly recommended to use that padding type because it is the most secure one. If you use the default PKCS #1 v1.5 padding, your ciphertext may be vulnerable to a Bleichenbacher padding oracle attack.

Let's check the files that we have created:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
2324953448 512 session_key.bin.encrypted
```

Note that the input session key file is only 32 bytes (256 bits) long, but its encrypted version is a whole 512 bytes (4,096 bits) long. It is because the RSA's output ciphertext block is the same size as the RSA key size used.

Try to encrypt several times and note that the encrypted file will have the same size but a different checksum after every encryption. Though RSA encryption *as such* is deterministic, the used OAEP padding is not deterministic, which leads to non-deterministic ciphertext.

We have successfully encrypted a small file. But what will happen if we try to encrypt a bigger file? Let's try. First, let's generate a bigger file:

```
$ seq 20000 >somefile.txt
```

We have got a file approximately 100 KB in size. Let's try to encrypt it:

```
$ openssl pkeyutl \
    -encrypt \
    -in somefile.txt \
    -out somefile.txt.encrypted \
    -pubin \
    -inkey rsa_public_key.pem \
    -pkeyopt rsa_padding_mode:oaep
Public Key operation error
C011FAE8677F0000:error:0200006E:
rsa routines:ossl_rsa_padding_add_PKCS1_OAEP_mgf1_ex:
data too large for key size:crypto/rsa/rsa_oaep.c:87:
```

We have got an error, telling us that our input data is too large for the key size. It is because RSA cannot encrypt data any longer than its key length during one operation. If padding is used, the maximum accepted plaintext length is even shorter. When OAEP padding is used, the maximum input data length is `key size - 42` – in our case, `4096 - 42 = 4054` bytes. If you need to encrypt longer data, you should encrypt it with a symmetric session key.

It's time to decrypt what we encrypted and compare it to the original. In order to decrypt, we need to provide the keypair to `openssl pkeyutl`, not just the public key:

```
$ openssl pkeyutl \
    -decrypt \
    -in session_key.bin.encrypted \
    -out session_key.bin.decrypted \
    -inkey rsa_keypair.pem \
    -pkeyopt rsa_padding_mode:oaep
```

Let's check the checksums:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
186975932 32 session_key.bin.decrypted
2324953448 512 session_key.bin.encrypted
```

As we can see, the sizes and checksums of `session_key.bin` and `session_key.bin.decrypted` match, meaning that the decrypted file is equal to the original file. It is absolutely great that decryption is always deterministic!

We have learned how to encrypt and decrypt on the command line. Now, let's learn how to encrypt with RSA programmatically.

How to encrypt with RSA programmatically

OpenSSL 3.0 provides the following APIs for RSA encryption:

- A legacy API with the `RSA_` prefix and the `RSA_public_encrypt()` function. This API has been deprecated since OpenSSL 3.0, so we are not going to use it.
- The `EVP_PKEY` API, particularly the `EVP_PKEY_encrypt()` function. We are going to use this API.
- The `EVP_Seal` API. This is a hybrid encryption API that generates a session key, encrypts the session key with RSA, and then encrypts the user data with the session key. This API contains the `EVP_SealInit()`, `EVP_SealUpdate()`, and `EVP_SealFinal()` functions, which work similarly to `EVP_EncryptInit()`, `EVP_EncryptUpdate()`, and `EVP_EncryptFinal()`. `EVP_SealUpdate()` is just `#define` for `EVP_EncryptUpdate()`. There are also the corresponding `EVP_Open` functions for decrypting the *seals* – `EVP_OpenInit()`, `EVP_OpenUpdate()`, and `EVP_OpenFinal()`. Unfortunately, the `EVP_Seal` API is rather inflexible. It supports only RSA encryption of the session key. The API does not provide a method to specify padding or other properties for RSA encryption, thus RSA encryption will always use the default PKCS #1 v1.5 padding instead of OAEP padding. It is a bit difficult to understand how to call the `EVP_SealInit()` function; you will likely spend several minutes looking at the documentation before you understand it. I think that the `EVP_Seal` API is badly designed in general and should be replaced by a better hybrid encryption API. Therefore, I do not recommend using the `EVP_Seal` API.

We are going to develop the `rsa-encrypt` program that encrypts a small portion of data – for example, a session key, using RSA, similar to how we encrypted with `openssl pkeyutl` in the previous section. Our program is going to be interoperable with `pkeyutl`, meaning that `pkeyutl` will be able to decrypt the ciphertext produced by our program.

As mentioned, we are going to use the `EVP_PKEY` API. Here is the list of the relevant man pages for the functions that we are going to use:

```
$ man PEM_read_PUBKEY
$ man EVP_PKEY_CTX_new_from_pkey
$ man EVP_PKEY_encrypt_init
$ man EVP_PKEY_CTX_set_rsa_padding
$ man EVP_PKEY_get_size
$ man EVP_PKEY_encrypt
$ man provider-asym_cipher
```

Our program will take three command-line arguments:

1. The input filename
2. The output filename
3. The RSA public key filename

Let's make a high-level implementation plan, as we usually do. Our program will need to perform the following steps:

1. Load the RSA public key from the RSA public key file.
2. Create the `EVP_PKEY` context from the key.
3. Initialize the `EVP_PKEY` context for encryption and set the OAEP padding mode.
4. Read the plaintext from the input file.
5. Encrypt the plaintext.
6. Write the produced ciphertext into the output file.

Implementing the rsa-encrypt program

Let's proceed with the implementation:

1. First, load the RSA public key:

```
const char* pkey_fname = argv[3];
FILE* pkey_file = fopen(pkey_fname, "rb");
EVP_PKEY* pkey = PEM_read_PUBKEY(
    pkey_file, NULL, NULL, NULL);
```

2. Next, create the EVP_PKEY context from the loaded key:

```
EVP_PKEY_CTX* ctx = EVP_PKEY_CTX_new_from_pkey(
    NULL, pkey, NULL);
```

3. Then, initialize EVP_PKEY_CTX and set the padding mode:

```
EVP_PKEY_encrypt_init(ctx);
EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_
PADDING);
```

You can also initialize in the new OpenSSL 3.0 style, using OSSL_PARAM with parameters:

```
int rsa_padding = RSA_PKCS1_OAEP_PADDING;
OSSL_PARAM params[] = {
    OSSL_PARAM_construct_int(
        OSSL_ASYM_CIPHER_PARAM_PAD_MODE, &rsa_padding),
    OSSL_PARAM_construct_end()
};
 EVP_PKEY_encrypt_init_ex(ctx, params);
```

It's interesting that the OpenSSL 3.0 documentation (on the provider-asym_cipher man page) defines the OSSL_ASYM_CIPHER_PARAM_PAD_MODE parameter as an integer, but in the OpenSSL source code (in the rsa_enc.c file), the same parameter is defined as a UTF-8 string. The code accepts both integer and string values and calls the legacy pad mode number integer value. When setting the padding mode with the string, the context initialization code will look like this:

```
OSSL_PARAM params[] = {
    OSSL_PARAM_construct_utf8_string(
        OSSL_ASYM_CIPHER_PARAM_PAD_MODE,
        OSSL_PKEY_RSA_PAD_MODE_OAEP,
        0),
    OSSL_PARAM_construct_end()
};
 EVP_PKEY_encrypt_init_ex(ctx, params);
```

I prefer the old initialization style because it's the most readable, concise, and type-safe, even though it is less generic. OpenSSL also prefers the old style. OpenSSL 3.0 source code contains some EVP_PKEY_CTX_set_rsa_padding() calls but no calls involving OSSL_ASYM_CIPHER_PARAM_PAD_MODE, except in the RSA implementation code.

4. The next thing we should do is read the input data. But how much should we read? We know that RSA with padding can encrypt slightly less than the `key size` bytes in one encryption operation. We can try to read and encrypt the `key size` bytes. If the encryption operation fails because the input was too long, we will discover it from the encryption error.

   Let's discover the key size, allocate input and output buffers of the appropriate size, and read the input data:

   ```
   size_t pkey_size = EVP_PKEY_get_size(pkey);
   unsigned char* in_buf  = malloc(pkey_size);
   unsigned char* out_buf = malloc(pkey_size);
   size_t in_nbytes = fread(in_buf, 1, pkey_size, in_file);
   ```

5. Asymmetric encryption of the input data is done using the `EVP_PKEY_encrypt()` function:

   ```
   size_t out_nbytes = pkey_size;
   EVP_PKEY_encrypt(
       ctx, out_buf, &out_nbytes, in_buf, in_nbytes);
   ```

6. The encrypted data should be written into the output file:

   ```
   fwrite(out_buf, 1, out_nbytes, out_file);
   ```

7. After the work is done, we have to deallocate the used buffers and objects:

   ```
   free(out_buf);
   free(in_buf);
   EVP_PKEY_CTX_free(ctx);
   EVP_PKEY_free(pkey);
   ```

The complete source code of our `rsa-encrypt` program can be found on GitHub as the `rsa-encrypt.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter06/rsa-encrypt.c`.

Running the rsa-encrypt program

Let's run the `rsa-encrypt` program and encrypt the 32-byte session key file that we created before:

```
$ ./rsa-encrypt \
    session_key.bin \
    session_key.bin.encrypted \
    rsa_public_key.pem
Encryption succeeded
```

Our program reported success; let's check the files:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
390548196 512 session_key.bin.encrypted
```

So far so good – our program has created the `session_key.bin.encrypted` file of the right size. I promised that our program would be interoperable with `openssl pkeyutl`. Let's check it:

```
$ openssl pkeyutl \
    -decrypt \
    -in session_key.bin.encrypted \
    -out session_key.bin.decrypted \
    -inkey rsa_keypair.pem \
    -pkeyopt rsa_padding_mode:oaep
```

No errors are reported.

Let's check the files and their checksums:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
186975932 32 session_key.bin.decrypted
2324953448 512 session_key.bin.encrypted
```

As we can observe, the sizes and checksums of `session_key.bin` and `session_key.bin.decrypted` match, meaning that `openssl pkeyutl` can successfully decrypt the file encrypted by the `rsa-encrypt` program.

I have mentioned error handling but have not shown any error handling code so far. I do so because I do not want the error handling code to clutter the code fragments, although the full complete source code on GitHub contains error handling. Error handling for OpenSSL is a topic in its own right and deserves a whole section in this chapter. Let's advance to this section, where we will also discuss the OpenSSL error queue.

Understanding the OpenSSL error queue

When running the `rsa-encrypt` program, many things can go wrong, such as the following:

- The public key file may be corrupted or not contain a key. In this case, key loading will fail.
- The public key file may contain a non-RSA key. In this case, key loading will succeed, but encryption will fail.
- The input file may be too big. In this case, encryption will also fail but for another reason.

How do we handle such errors? Those OpenSSL functions that can fail usually indicate so by returning `NULL`, `0`, or a negative number. Success is usually indicated by returning `1`. Some functions also add an error to the OpenSSL error queue on failure.

The OpenSSL error queue is a container for errors that the OpenSSL library wants to report. *Every thread of the process has its own OpenSSL error queue.* The error queue does not require initialization or uninitialization; OpenSSL automatically handles it. Every thread starts with an empty error queue. The error queue stores errors until they are taken out of the queue or the thread finishes execution. Therefore, it is a good idea to check the error queue, handle the errors, and if needed, clear the queue after calls to OpenSSL.

Not all functions put errors into the error queue on failure – for example, I was unable to get any errors into the OpenSSL error queue on symmetric decryption failure and HMAC calculation initialization failure. But OpenSSL functions handling asymmetric cryptography, X.509 certificates, or TLS usually support the error queue and put errors there on failure. Some functions, especially those that verify X.509 certificates, can put several errors into the error queue during one call.

Here are some popular functions for interacting with the OpenSSL error queue:

- `ERR_get_error()`: Get the code of the earliest error in the queue and remove that error from the queue.
- `ERR_peek_error()`: Get the code of the earliest error in the queue but leave that error in the queue.
- `ERR_GET_LIB()` and `ERR_GET_REASON()`: Get components of the code returned by `ERR_get_error()` or `ERR_peek_error()`.
- `ERR_error_string_n()`: Make a human-readable error string from the error code.
- `ERR_clear_error()`: Clear the queue and remove all the errors from it.
- `ERR_print_errors_fp()`: Print the error queue to a `FILE` stream and clear the queue. Note that while this function is very convenient for debugging, it cannot indicate failure if it fails to print the error queue – for example, if the `FILE` stream writes to a filesystem and the disk is full. The `ERR_print_errors_fp()` function neither returns an error code nor puts an error into the error queue because its task is to also clear the queue.
- I find this failure to report failures in a failure handling function a bit ironic, even though I find the behavior of this function logical.

It is important to understand that exit codes returned by OpenSSL functions are not the same as error codes contained in the errors that are put into the error queue – for example, if input for RSA encryption is too long, the `EVP_PKEY_encrypt()` function returns the `0` *exit code* and puts an error into the error queue with the `0x200006E` *error code*, which consists of the following components:

- A library number provided by `ERR_GET_LIB()`: `4`, which is `ERR_R_RSA_LIB`

- A reason code provided by `ERR_GET_REASON()`: `0x6E`, which is `110`, which is `RSA_R_DATA_TOO_LARGE_FOR_KEY_SIZE`

Here is sample code for getting the earliest error from the error queue and printing the error information:

```
unsigned long error_code = ERR_get_error();
printf("Error code: %lX\n", error_code);
printf("Library number: %i\n", ERR_GET_LIB(error_code));
printf("Reason code: %X\n", ERR_GET_REASON(error_code));
```

The sample code provided will print the library number and the reason code as numbers. But where can we find their symbolic representations, such as `ERR_R_RSA_LIB` and `RSA_R_DATA_TOO_LARGE_FOR_KEY_SIZE`? Unfortunately, I could not find the answer in the OpenSSL documentation, but I could find the answer in the OpenSSL library headers. The list of library numbers can be found in the `err.h` header, and the list of library-specific reason codes can be found in the library-specific error header file – for example, for the RSA sub-library, the library-specific error header file is `rsaerr.h`.

The `ERR_get_error()`, `ERR_peek_error()`, `ERR_GET_LIB()`, and `ERR_GET_REASON()` functions provide a way for sophisticated error handling of OpenSSL calls. But what if we want something quick and easy for debugging? Then, we can use the `ERR_print_errors_fp()` function. We can do it just like this:

```
ERR_print_errors_fp(stderr);
```

The error queue will be printed to `stderr` and cleared.

That's how the `ERR_print_errors_fp()` function is used in the source code of our `rsa-encrypt` program:

```
if (ERR_peek_error()) {
    exit_code = 1;
    if (error_stream) {
        fprintf(
            error_stream,
            "Errors from the OpenSSL error queue:\n");
        ERR_print_errors_fp(error_stream);
    }
}
```

Let's try to run `rsa-encrypt` with an input that is too long:

```
$ seq 20000 >somefile.txt
$ ./rsa-encrypt \
    somefile.txt \
    somefile.txt.encrypted \
    rsa_public_key.pem
EVP_API error
Errors from the OpenSSL error queue:
806B3E2A797F0000:error:
  0200006E:rsa routines:
  ossl_rsa_padding_add_PKCS1_OAEP_mgf1_ex:
  data too large for key size:
  crypto/rsa/rsa_oaep.c:87:
Encryption failed
```

We have got the same printout from the OpenSSL error queue that `openssl pkeyutl` prints on the same error. There is a lot of information for one error, but some part of the long error string is human-readable:

```
  data too large for key size
```

You can find more information on OpenSSL call error handling on the OpenSSL man pages:

```
$ man ERR_get_error
$ man ERR_GET_LIB
$ man ERR_error_string_n
$ man ERR_print_errors_fp
$ man ERR_clear_error
```

It is, of course, up to you how you are going to handle errors from the OpenSSL calls. But as a responsible programmer, you should not forget to process and clear the OpenSSL error queue after failures.

Imagine that you performed several operations with OpenSSL. For most operations, function exit codes were enough for you, and you forgot to clear the error queue, but for the last failed operation, you want to get the error from the queue. The queue, however, now contains errors from all your performed operations. How are you going to find the errors for the last operation among all the other errors? There is no good way. Therefore, you should start your operation with an empty error queue. By *operation* here, I don't necessarily mean a single call to OpenSSL. An operation may contain several calls, but it should represent a logical piece of work so that you are comfortable with processing all the operation's errors at once.

When is it better to clear the OpenSSL error queue – before or after the operation? Different people have different opinions on it. One opinion is that the error queue should be cleared after the operation because a responsible programmer should clean after themselves and not *leak* errors. Another opinion is that clearing the error queue before the operation is better because it ensures an empty error queue before the operation. I prefer to clear the queue both before and after the operation – after because it is responsible, and before because in complex projects where many people are contributing, one or more persons will sometimes forget to clear the error queue after themselves. Humans make mistakes; it's the sad truth of life and software development.

Alright, enough with philosophy. It's now time to learn how to decrypt with RSA using C code.

How to decrypt with RSA programmatically

In this section, we are going to develop a small rsa-decrypt program, so that we can learn how to do RSA decryption.

Our program will take three command-line arguments similar to those taken by `rsa-encrypt`, but note that the third argument is the name of a file containing a keypair, not just a public key:

1. The input filename
2. The output filename
3. The RSA keypair filename

Of course, now the input file is expected to contain ciphertext, which will be decrypted, and the resulting plaintext will be written into the output file.

Our high-level implementation plan for `rsa-decrypt` will contain the sequence of actions opposite to that of `rsa-encrypt`, listed as follows:

1. Load an RSA keypair from the RSA keypair file. We need an RSA private key for decryption; a public key will not be enough.
2. Create the `EVP_PKEY` context from the key.
3. Initialize the `EVP_PKEY` context for decryption and set the OAEP padding mode.
4. Read ciphertext from the input file.
5. Decrypt the ciphertext.
6. Write the produced plaintext into the output file.

Implementing the rsa-decrypt program

Let's implement the `rsa-decrypt` program according to our plan, step by step:

1. First, load the RSA keypair:

```
const char* pkey_fname = argv[3];
FILE* pkey_file = fopen(pkey_fname, "rb");
EVP_PKEY* pkey = PEM_read_PrivateKey(
    pkey_file, NULL, NULL, NULL);
```

   Note that the same type, `EVP_PKEY`, is used for holding either a public key or a whole keypair.

2. Next, create the `EVP_PKEY` context from the loaded keypair:

```
EVP_PKEY_CTX* ctx = EVP_PKEY_CTX_new_from_pkey(
    NULL, pkey, NULL);
```

3. Then, initialize `EVP_PKEY_CTX` and set the padding mode:

```
EVP_PKEY_decrypt_init(ctx);
EVP_PKEY_CTX_set_rsa_padding(ctx, RSA_PKCS1_OAEP_
PADDING);
```

4. Find out the key size, allocate input and output buffers of the appropriate size, and read the input data:

```
size_t pkey_size = EVP_PKEY_get_size(pkey);
unsigned char* in_buf  = malloc(pkey_size);
unsigned char* out_buf = malloc(pkey_size);
size_t in_nbytes = fread(in_buf, 1, pkey_size, in_file);
```

5. Decrypt the obtained ciphertext using the `EVP_PKEY_decrypt()` function:

```
size_t out_nbytes = pkey_size;
EVP_PKEY_decrypt(
    ctx, out_buf, &out_nbytes, in_buf, in_nbytes);
```

6. Write the decrypted data into the output file:

```
fwrite(out_buf, 1, out_nbytes, out_file);
```

7. After the work is done, we have to deallocate the used buffers and objects:

```
free(out_buf);
free(in_buf);
```

```
    EVP_PKEY_CTX_free(ctx);
    EVP_PKEY_free(pkey);
```

The complete source code of our `rsa-decrypt` program can be found on GitHub as the `rsa-decrypt.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter06/rsa-decrypt.c`.

Running the rsa-decrypt program

Let's run the `rsa-decrypt` program and decrypt the previous encrypted session key file that we created before:

```
$ ./rsa-decrypt \
    session_key.bin.encrypted \
    session_key.bin.decrypted \
    rsa_keypair.pem
Decryption succeeded
```

Our program reported success. Let's check the files and their checksums:

```
$ cksum session_key.bin*
186975932 32 session_key.bin
186975932 32 session_key.bin.decrypted
2324953448 512 session_key.bin.encrypted
```

As we can see, the sizes and checksums of `session_key.bin` and `session_key.bin.decrypted` match, meaning that our `rsa-decrypt` can successfully decrypt the session key file.

Summary

In this chapter, we learned about the concept of asymmetric cryptography. Then, we learned about the concept of an MITM attack and also how to mitigate that attack. After that, we learned what a session key is. We finished the theoretical part by learning about RSA security, the trade-off between security and performance, and what RSA key size to choose.

In the practical part, we learned how to encrypt and decrypt with RSA on the command line. Then, we learned how to encrypt and decrypt with RSA programmatically in C code. We also learned about the OpenSSL error queue and how to check it for errors.

In the next chapter, we will continue learning about asymmetric cryptography. Particularly, we will learn about digital signatures and their verification.